VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

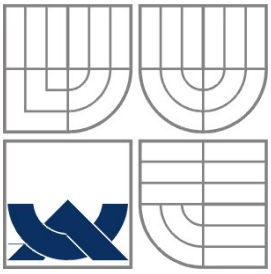
JAZYK PRO POPIS INSTRUKČNÍCH SAD

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

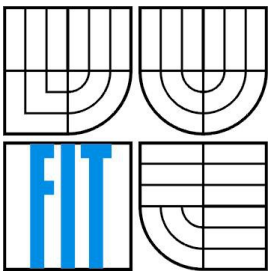
AUTOR PRÁCE
AUTHOR

JAN FOREJTNÍK

BRNO 2014



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

JAZYK PRO POPIS INSTRUKČNÍCH SAD

A LANGUAGE FOR DESCRIPTION OF INSTRUCTION SETS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JAN FOREJTNÍK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. ALEŠ SMRČKA, PhD.

BRNO 2014

Abstrakt

V této práci je představen návrh jednoduchého jazyka pro popis architektury mikroprocesoru zaměřeného na popis instrukční sady. Dále je popsána implementace interpretu tohoto jazyka, který je schopen simulovat popsanou architekturu. Tato práce může zároveň sloužit jako návod k používání tohoto interpretu.

Abstract

This bachelor's thesis introduces a simple concept of a language for description of microprocessor architecture, namely the instruction set. An interpreter of the language capable of simulating the behavior of the architecture is briefly described. This text may also serve as a manual for using the interpreter.

Klíčová slova

Instrukční sada, CPU, počítačová architektura, jazyk, simulátor, ADL

Keywords

Instruction set, CPU, computer architecture, language, simulator, ADL

Citace

Jan Forejtník: Jazyk pro popis instrukčních sad,
bakalářská práce, Brno, FIT VUT v Brně, 2014

Jazyk pro popis instrukčních sad

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Aleše Smrčky, PhD.

Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . . . .
Jan Forejtník
20. května 2014

Poděkování

Zde děkuji především mému vedoucímu, Ing. Aleši Smrčkovi, za vždy přátelské a přínosné konzultace vztahující se k této práci.

Obsah

1 Úvod

V minulosti se návrh a testování hardwarových komponent prováděly manuálně. Z logických hradel byly vytvářeny prototypy obvodů, jejichž funkčnost byla testována na nepájivých polích. S překotným technologickým vývojem a nutností vyrábět mnohem složitější obvody se začalo uvažovat, jak proces vývoje, testování a simulace alespoň částečně automatizovat. To byla hlavní motivace pro vznik jazyků pro popis hardware (dále jen *HDL*). Tyto jazyky umožňují dostatečně přesný formální popis struktury a chování hardware, na jehož základě je možno nejen provádět automatickou analýzu, simulaci a verifikaci, ale i přímo navrhovat fyzický hardware (proces označovaný jako *syntéza*). Mezi nejznámější HDL jazyky patří *VHDL* [1] či *Verilog* [2].

Ne vždy však potřebujeme popisovat hardwarovou architekturu na takto nízké úrovni. Naopak, současným trendem při vývoji hardware jsou v dnešní době především metodologie popisující hardware na abstraktnější, systémové úrovni často označované termínem *Electronic System-Level design* [3]. Jedním z prostředků pro tuto metodologii je popis pomocí *Architecture Description Languages* [4] (dále jen *ADL*). Jelikož termín ADL je poměrně široký a krom výše zmíněného popisu hardwarových architektur je možno jej použít i pro popis softwaru nebo dokonce i pro popis business procesů [5], tímto deklaruji, že všechny výskyty termínu ADL v této práci se vztahují k jazykům pro popis počítačových architektur, protože ty jsou hlavním tématem celé práce.

Mým úkolem bylo vytvořit jednoduchý, dostatečně obecný ADL pro popis instrukční sady procesoru a implementovat interpret tohoto jazyka specializovaný na sémantiku instrukcí, který bude sloužit jako podpůrný nástroj pro formální verifikaci architektur. Kroky popisující jednotlivé fáze vývoje i jeho výsledky jsou popsány v jednotlivých kapitolách tohoto textu.

V kapitole 2 popisuji třídy existujících ADL, představuji několik zástupců těchto jazyků a shrnuji jejich vlastnosti a schopnosti. Na základě analýzy nejen těchto existujících jazyků shrnuji v kapitole 3 požadavky nutné pro dostatečně obecný, jednoduchý ADL. Kapitola 4 přestavuje koncept navrženého jazyka vysvětlený na praktických příkladech. Kapitola 5 popisuje mnou vytvořenou implementaci tohoto jazyka ve formě interpretu. Schopnosti a příklady použití tohoto interpretu demonstruji v kapitole 6 na zvolených příkladech instrukčních sad.

2 Existující jazyky

V této kapitole se pokusím představit některé z již existujících ADL. Tyto jazyky můžeme, podle terminologie uvedené v [4], z hlediska informací, které se snaží zachytit, rozdělit do několika tříd (přičemž třídy nejsou exkluzivní a jeden jazyk může být ve více třídách současně).

Strukturální jazyky detailně popisují rozhraní jednotlivých komponent a jejich propojení. *Behaviorální jazyky* popisují především instrukční sadu a účinky jednotlivých instrukcí, jakými jsou např. čtení či zápis v rámci paměťových elementů. *Dílčí jazyky* popisují architektury ze značně specifického hlediska, které nelze zařadit to předešlých dvou kategorií. Jazyky, které kombinují strukturální a behaviorální přístup, označujeme jako *smíšené jazyky*.

Jako příklady budu uvádět především behaviorální jazyky, neboť do této kategorie spadá i jazyk, který budu navrhovat.

2.1 SystemC

SystemC [6] je sada knihoven (přesto je ale často nazýván samostatným jazykem) programovacího jazyka C++, která zajišťuje podporu pro modelování hardwarových architektur. SystemC lze považovat zároveň za ADL a HDL, neboť obsahuje prostředky pro popis na více úrovní abstrakce. Konkrétně jde např. o podporu *cycle-accurate simulation* (simulace na úrovni cyklů, dále jen *CAS*), vícevláknové prostředí konkurentních procesů, časovače, množinu architekturálně specifických událostí a mnohé další. Pomocí prostředků SystemC lze generovat simulátory na mnoha úrovních abstrakce od funkcionální až po kompletně časovanou simulaci.

SystemC je standardizovaný simulační jazyk a proto jej umí zpracovat většina simulátorů počítačových architektur. SystemC je již léta používán v komerčním sektoru, využívá jej např. firma Synopsys [7], která stála i u zrodu tohoto projektu.

2.2 ArchC

ArchC [8] je open-source ADL, jehož vývoj započal na brazilské University of Campinas již v roce 2003. ArchC syntakticky vychází z jazyka C, potažmo SystemC. Spadá do kategorie smíšených ADL, neboť vyžaduje strukturální i behaviorální popis.

Zdrojový kód jazyka je rozdělen do dvou částí. Část *Instruction Set Architecture* popisuje formát, velikost, název instrukcí a všechny informace potřebné pro dekódování a také vykonávání instrukce. Druhou částí je *Architecture Resources*, která popisuje úložiště dat, jako jsou registry nebo asociativní paměť, a strukturu zřetězeného zpracování.

Krom samotného jazyka poskytují vývojáři ArchC také sadu nástrojů jako je simulátor, překladač nebo asembler. Jelikož jazyk podporuje více úroví abstrakce, na základě informací ze zdrojového kódu je možno spouštět jak simulaci na úrovni instrukcí, tak na úrovni cyklů procesoru. Součástí ArchC toolkitu je i SystemC simulátor.

2.3 LISA

Původní verze jazyku *LISA* (Language for Instruction Set Architecture) [9] vznikla na půdě RWTH Aachen University již v roce 1996. V článku [10] byla v roce 1999 představena nová verze jazyka, která je schopna popisovat chování architektury na úrovni cyklů. Jazyk spadá do kategorie smíšených ADL. Soustředí se na tzv. *behavioral pipelining*, při kterém dokáže detailně popsat jednotlivé fáze zřetězeného zpracování instrukcí a jednotlivé operace mohou být přidruženy k určité fázi zřetězeného zpracování.

Tak jako všechny předešlé jazyky definuje nejprve zdroje, kterými jsou registry, paměť, zřetězené zpracování či sběrnice, a poté operace nad těmito zdroji. Jazyk popisuje architekturu v dostatečném detailu k tomu, aby ze zdrojového kódu mohla být provedena syntéza hardware.

3 Analýza požadavků na jazyk

V této kapitole uvádím požadavky na navrhovaný jazyk, které jsem získal analýzou existujících jazyků a také na základě konzultace s vedoucím práce. Navrhovaný jazyk by měl patřit do třídy behaviorálních ADL a měl by se soustřeďovat především na popis sémantiky instrukcí. Přidanou hodnotou oproti existujícím jazykům by měla být jeho jednoduchost, kterou většina výše zmíněných jazyků postrádá.

3.1 Délka instrukcí

Každá instrukce zabírá v paměti počítače paměťový prostor o určité velikosti. Z tohoto hlediska dělíme instrukční sady na sady s pevnou nebo proměnlivou délkou instrukcí.

Instrukční sady s pevnou délkou instrukcí mají takové instrukční sady, ve kterých každá instrukce zaujímá v paměti prostor o stejné velikosti. Výhodou takové architektury je, že instrukční dekodér předem zná délku současné (a vzhledem k tomu, že všechny instrukce mají stejnou délku, i následující) instrukce a to zjednodušuje proces dekódování. Krom toho je tímto přístupem zjednodušeno případné zřetězené zpracování instrukcí. Naopak nevýhoda tohoto přístupu je snížení hustoty kódu, neboť ne vždy daná instrukce naplno využívá paměťový prostor, který zabírá. Pevnou délku instrukcí používá většina RISC architektur. Pomineme-li spoustu dnes již archaických architektur, z těch novějších lze jmenovat např. *ARM* [11].

Druhou skupinou jsou architektury s proměnlivou délkou instrukcí. Jednotlivé instrukce se mohou, co se týče jejich délky, výrazně lišit. Výhody a nevýhody jsou komplementární vůči architekturám s pevnou délkou instrukcí. Díky mnohdy velice krátkým instrukcím je docílena vysoká hustota kódu. Naopak kvůli proměnlivé délce instrukcí je potřeba výrazně složitějšího dekodéru a často také vícenásobných přístupů do paměti, které zpomalují běh systému. Proměnlivou délku instrukcí používá např. architektura *Intel x86*, jejíž instrukce mohou mít délku od 1B do 15B [12].

Jelikož navrhovaný jazyk má být dostatečně obecný, měl by být schopen popsat instrukční sady s pevnou i s proměnlivou délkou instrukcí.

3.2 Binární podoba instrukcí

Jednotlivé instrukce jsou v paměti reprezentovány posloupností hodnot 1 nebo 0 (dále jen *binární instrukce*) o určité délce. Aby bylo možné dvě instrukce navzájem rozlišit, musí se hodnoty jejich bitů alespoň na jedné pozici lišit. Proto je každá instrukce v instrukční sadě jednoznačně identifikována svým operačním kódem.

Operačním kódem instrukce rozumíme ty bity binárního vektoru, na jejichž hodnotu se dotazuje instrukční dekodér během dekódování instrukce. Definujeme-li tedy operační kód instrukce, určujeme které bity posloupnosti mají jakou hodnotu. Aby byly všechny instrukce v instrukční sadě jednoznačně rozlišitelné, musí se operační kód každé dvojice instrukcí z této instrukční sady lišit hodnotou alespoň jednoho bitu.

Operand je jeden nebo více bitů binární instrukce, na jehož hodnotu není brán ohled při dekódování instrukce. Jelikož jsou operandy téměř vždy nositelem informace potřebné pro vykonávání instrukce, bylo by vhodné, aby navrhovaný jazyk poskytoval možnost se na hodnoty jednotlivých operandů odkazovat.

Binární podoba instrukce je popsána operačním kódem a operandy. Jelikž v reálně používaných architekturách operační kód ani operandy instrukce často nejsou souvislým blokem po sobě jdoucích bitů, proto by navrhovaný jazyk měl mít schopnost definovat jakékoliv rozložení operačního kódu a operandů v binární instrukci. Pro popis binární podoby instrukce v dalších částech této práce zavádím notaci, ve které je instrukce reprezentována řetězcem složeného ze znaků `0`, `1` a `?`, kde `0` a `1` jsou hodnoty operačního kódu na dané pozici, zatímco `?` indikuje, že na dané pozici je operand. Nejméně významný bit v tomto řetězci je ten nejpravější.

3.3 Popis účinků vykonání instrukce

U každé instrukce je nutné popsat popis účinků vykonání této instrukce. Jedná se posloupnost příkazů k přesunu dat mezi paměťovými entitami v rámci architektury. Pro vyšší pohodlnost a zamezení zbytečné redundance kódu v popisu chování instrukcí by měl navrhovaný jazyk zahrnovat také definici a volání funkcí a lokální proměnné.

3.4 Popis zdrojů

Každá architektura disponuje určitým počtem paměťových zdrojů jako je registr, registrové pole či asociativní paměť (dále jen *paměťové entity*). Jelikož u každé z těchto entit je třeba specifikovat jejich velikost, musí navrhovaný jazyk být schopen definovat bitovou šířku každého registru, počet registrů v registrovém poli a šířku sběrnice, velikost buňky a počet buněk asociativní paměti. V drtivé většině současných architektur odpovídají bitové šířky registrů a buněk paměti standardním datovým typům známým z vyšších programovacích jazyků (násobky jednoho bajtu). Dostatečně obecný jazyk by však v tomto směru neměl klást žádná omezení, proto je žádoucí, aby bitové šířky byly zadávány přímo v jednotkách bitů.

3.5 Výrazové schopnsoti

Při čtení (zápisu) z (do) paměťových entit dochází k manipulaci s binárními vektory různých délek a hodnot. Jazyk by měl mít dostatečnou výrazovou vyjadřovací sílu k tomu, aby dokázal selektovat jakoukoliv část libovolného binárního vektoru a to včetně převracení pořadí významových bitů v bloku bitů či spojování více nesouvislých částí do jedné. Dále je nad binárními vektory nutná podpora elementárních aritmetických a logických operací. Těmi jsou:

- sčítání
- odčítání
- násobení
- výsledek celočíselného dělení
- zbytek po celočíselném dělení
- bitový posun vlevo
- logický posun vpravo
- aritmetický posun vpravo
- doplněk (NOT)
- bitový součet (OR)
- bitový součin (AND)
- bitová nonekvivalence (XOR)

4 Návrh jazyka

Po konzultaci s vedoucím práce jsem se při tvorbě gramatiky jazyka rozhodl vycházet ze syntaxe programovacího jazyka C. Při návrhu jazyka byly zohledněny požadavky z kapitoly 3. V této kapitole jsem se rozhodl, namísto formální definice, popsat syntax i sémantiku jazyka na konkrétním demonstrativním příkladu. Zájemce o vyčerpávající popis formální definice jazyka, popsané pomocí BNF notace, odkazuji do přílohy A.

4.1 Demonstrativní příklad architektury

V této podkapitole budu na vhodně zvoleném příkladu smyšlené architektury demonstrovat vlastnosti jazyka. Pro lepší názornost bude příklad strukturován do bloků zdrojového kódu s následným komenářem odkazujícím se do něj. Ve zdrojovém kódu budou barevně rozlišeny `terminální symboly` (klíčová slova jazyka), `identifikátory` a `číselné konstanty`.

```
1  architecture x69
2  {
3     fetch_size = 0b1000;
4     const four = 4;
5     const twenty_three = 0x17;
6  };
```

V tomto bloku kódu definujeme entitu typu `architecure` nazvanou `x69`. Ačkoliv syntax jazyka dovoluje, aby se entita tohoto typu ve zdrojovém kódu libovolněkrát (dokonce i vůbec), z praktického hlediska to nedává příliš smysl a ve většině případů je optimální právě jeden výskyt. Ve svém těle, tj. mezi znaky `{` a `}`, může obsahovat libovolný počet proměnných nebo konstant. V tomto případě je to proměnná `fetch_size` a konstanty `four` a `twenty_three`. Proměnné **mohou** mít definované počáteční hodnoty. Konstanty **musí** mít definované počáteční hodnoty. Číselné konstanty lze vyjádřit v binární (3), dekadické (4) nebo hexadecimální podobě (5). Entita tohoto typu je typicky využita právě k ukládání globálně přístupných architekturálně specifických konstant, ale jelikož z obecného hlediska nenese žádnou informaci nutnou k popisu architektury, může být zcela vynechána.

```
7   register pc(8);
8   register acc(four);
9   register arr[four](x69.fetch_size);
```

Na řádku 7 definujeme entitu typu `register` nazvanou `acc`. Zápis `(8)` říká, že bitová šířka tohoto registru je 8 bitů. Na řádku 9 definujeme entitu typu `register_array`, kde počet registrů v registrovém poli je dán hodnotou konstanty `four` a bitová šířka každého z registrů odpovídá hodnotě proměnné `x69.fetch_size`.

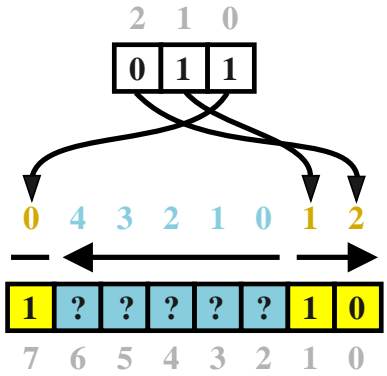
```
10  memory ram(5, x69.fetch_size);
11  memory cache(3, 12)
12  {
13     cache_enabled = 1;
14  };
```

Řádky 10-14 definují entity typu `memory`, tedy asociativní paměť. Definujeme-li paměť, uvádíme dva parametry. První parametr specifikuje šířku adresové sběrnice paměti. Jelikož v případě paměti `ram` je to číslo 5, jsou validní adresy pro přístup do této paměti v rozmezí 0 až $2^5 - 1$. Druhým parametrem je šířka datové sběrnice, respektive velikost jedné buňky paměti. V případě paměti `ram` je to hodnota proměnné `x69.fetch_size`. Volitelně lze u paměti, a stejně tak u všech ostatních paměťových entit, přidat blok deklarující proměnné či konstanty (12–14).

```
15  ram.read(addr)
16  {
17     // handle cache miss
18     if (cache[addr[0..1]] == addr[2..3])
19     {
20        cache[addr[0..1]] = addr[2..3] @ ram[addr];
21     }
22     else {} // pouze pro demonstraci
23     return cache[addr[0..1]][2..9];
24  }
```

Z čistě deklarativního popisu se přesouváme do imperativního popisu chování. Na řádcích 15-27 definujeme funkci `read` přidružené k entitě `ram`, která zpracovává jeden parametr `addr`. Kromě zobrazených řídících konstrukcí `if`, `if-else` a `return` podporuje jazyk i řídící cyklus `while`, který má totožnou syntax jako v jazyku C. Jako operátory porovnávacích výrazů mohou sloužit symboly `==`, `!=`, `<`, `<=`, `>`, `>=` jejichž význam je na první pohled zřejmý. Konstrukce `addr[0..1]` je řez, tedy výběr podmnožiny výrazu. Binární operátor `@` (20) slouží ke konkatenaci dvou výrazů.

```
25  instruction LOAD
26  (
27      [7]@[1..0] == 3,
28      addr = [2..6]
29  )
30  {
31      temp abc = ram.read(addr);
32      acc = abc;
33      pc = pc + 1;
34  }
```



Obrázek 4.1: Binární podoba instrukce LOAD

Blok 25-34 definuje instrukci LOAD. V hlavičce instrukce (26-29) definujeme binární podobu instrukce pomocí alespoň jednoho bloku operačního kódu a libovolného počtu operandů. Operační kód (27) se vyskytuje na pozicích definovaných regionem [7]@[1..0], není pojmenován a hodnota tohoto operačního kódu je 3. Všimněme si, že v části regionu [1..0] je, oproti všem předchozím případům, levý index vyšší než pravý. Tím je v daném regionu docíleno převrácení pořadí významových bitů. Operand addr (28) je v pozici definované regionem [2..6]. Celou hlavičku, a tedy podobu binární instrukce, ilustruje obrázek 4.1. Šipky a nad nimi barevně zvýrazněné indexy jednotlivých bitů znázorňují směr růstu významnosti bitů v daném regionu. V horní části obrázku je vidět distribuci hodnoty 3 do regionu [7]@[1..0].

V těle (30-34) popisujeme účinky instrukce jako sekvenci příkazů. Řádek 31 ukazuje definici lokální proměnné abc. Lokální proměnné jsou výhodné pro zkracování zápisu mnohdy často dlouhých výrazů a lze je použít v těle funkcí i instrukcí. Příkazy, výrazy a jejich sémantika jsou popsány v 4.3.

```
34  reset x69.behavior()
35  {
36      pc = 0; // likely to be overriden by INIT()
37      INIT();
38      while (RUN)
39      {
40          ir = ram.read(pc);
41          execute(ir);
42      }
43  }
```

V posledním bloku demonstrativního kódu vidíme funkci, jejíž klasické definici předchází klíčové slovo `reset`. Tato funkce se ve zdrojovém kódu musí nacházet právě jednou. Jedná se o vstupní bod chování architektury a vyjadřuje stav bezprostředně po uvedení architektury do chodu.

`INIT()` (37) je klíčové slovo, pod kterým se skrývá vestavěná funkce jazyka, která nastavuje počáteční podmínky (tj. hodnoty proměnných a paměťových prvků). Z formálního jazykového hlediska je tato funkce zcela nepodstatná a lze ji ignorovat. Funkce `INIT()` byla do jazyka přidána čistě z praktických implementačních důvodů, které jsou popsány v 5.8.6. Tuto konstrukci lze, jakkoli je to nepraktické, nahradit sekvencí přiřazovacích příkazů (36).

Výraz `RUN` (38) je vestavěný výraz jazyka, který zastupuje obecnou podmínku pro ukončení vykonávání chování architektury. Z formálního hlediska jde o konstantu, která je vždy ohodnocena jako pravda. Praktické využití je popsáno v 5.8.10.

Funkce `execute` (41) je vestavěná funkce jazyka, jejíž úkolem je dekódovat instrukce a následně provádět jejich tělo (výstižnější název by patrně byl `decode_and_execute`, ale pro větší pohodlnost programátora byl použit zkrácený název). Tato funkce existuje ve dvou odlišných signaturách – s jedním a se dvěma parametry. V případě jediného parametru je tento argument vyhodnocen a jeho hodnota (binární vektor) je použita k pokusu o dekódování instrukce. Druhou variantou je tvar `execute(ram, addr)`, kde prvním parametrem je název paměťové entity, ze které hodláme instrukci číst, a druhým parametrem je adresa, na které v dané paměťové entitě hodláme začít dekódovat instrukci. Tato varianta byla do jazyka přidána kvůli architekturám s proměnlivou délkou instrukcí, kdy hrozí možnost, že instrukce je delší než hodnota prvotně přečtená z paměti. Více informací o praktické stránce dekódování je v 5.9.

4.2 Datové typy

V jazyku se vyskytují dva druhy datových typů. Prvním, elementárním typem je *výraz* (popsaný v 4.3). Druhým typem jsou strukturované datové typy (dále jen *entity*) `architecture`, `instruction`, `memory`, `register`, `register_array`.

Všechny entity mohou nést libovolný počet proměnných či konstant. Entity `architecture` a `instruction` jsou specifické tím, že narozdíl od ostatních entit nenesou vlastní hodnotu a nelze je proto samostatně (tj. pouze pomocí identifikátoru) použít ve výrazech. Entity `memory` a `register_array` lze použít jako výraz jenom v případě, vyskytuje-li se bezprostředně za ní operátor řezu, který specifikuje, která buňka paměti, resp. který konkrétní registr v poli, se použije jako hodnota výrazu.

4.3 Výrazy a operátory

Za každým výrazem jazyka se skrývá bitový vektor jakožto jediný elementární datový typ. Výrazem jazyka je jedna z následujících variant:

- proměnná nebo konstanta přidružená k entitě (`x69.fetch_size`)
- lokální proměnná (`abc`)
- hodnota registru (`acc`, `arr[2]`)
- hodnota jedné či více buněk paměti (`ram[5]`, `ram[5..6]`)
- návratová hodnota funkce (`ram.read(addr)`)
- výsledek binární či unární operace (`abc + ram[5]`, `-acc`) (dále popsáno podrobněji)

4.3.1 Unární operace

Jayzk obsahuje dvě unární operace:

- řez
  - `0b100[0] == 0b0`
  - `0b101[0..2] == 0b101`
  - `0b100[2..0] == 0b001` (změna orientace významových bitů)
  - `0b11110000[1..6][4..1] == 0b0011` (vícenásobný řez)
  - `0b10001011[0..2]@[5..3] == 0b100011` (konkatenace řezů)
  - `0b101[0..2]@[0..2]@[0..2] == 0b101101101`
- negace
  - `-0b101 = 0b010`

4.3.2 Binární operace

Všechny binární operace jsou v jazyce zapsány ve formátu `první_operand operátor druhý_operand`. Binární operace jsou všechny zleva asociativní a nemají definované precedence. K dispozici jsou následující operace, resp. `operátory`:

- sčítání
  - `0b101 + 0b110 == 0b1011`
- odečítání
  - `0b101 - 0b110 == 0b111`
- násobení
  - `0b101 * 0b110 == 0b11110`
- celočíselné dělení kladných čísel
  - `0b101 / 0b110 == 0b000`
- zbytek po celočíselném dělení kladných čísel
  - `0b101 % 0b110 == 0b101`
- bitový posun vlevo
  - `0b101 << 0b10 == 0b10100`
- logický posun vpravo
  - `0b101 >> 0b10 == 0b001`
- aritmetický posun vpravo
  - `0b101 ~> 0b10 == 0b111`
- bitový součet
  - `0b101 | 0b100 == 0b101`
- bitový součin
  - `0b101 & 0b100 == 0b100`
- bitová nonekvivalence
  - `0b101 ^ 0b100 == 0b001`
- konkatenace
  - `0b101 @ 0b100 == 0b100101`

- porovnávací operátory `==`, `!=`, `<`, `<=`, `>`, `>=`
  - na rozdíl od předešlých operací není výsledek porovnání přiřaditelný výraz

5 Implementace interpretu jazyka

5.1 Použité technologie

Interpret je implementován v jazyce C++ a využívá i některé exkluzivní vlastnosti z jeho nejnovější normy C++11, tudíž je podpora této normy nutnou podmínkou k úspěšnému překladu aplikace.

K tvorbě lexikálních analyzátorů byl použit automatický generátor `flexc++` [X]. Syntaktické analyzátory byly vygenerovány nástrojem `bisonc++` [X]. Jedná se patrně o jedinou dnes (2014-5) zdarma dostupnou dvojici generátor lexikálních analyzátorů, generátor syntaktických analyzátorů, která používá čistě objektově orientované C++ rozhraní. Autora těchto knihoven, kterým je Frank B. Brokken, jsem dokonce během vývoje aplikace kontaktoval. Bohužel pro mě se chyba, kterou jsem mu v e-mailu popisoval, ukázala být naopak žádanou vlastností nástroje `flexc++`. Jednou z velkých nevýhod nástroje `bisonc++` je obtížné zotavování ze syntaktických chyb. Kvůli tomu byl do aplikace přidán specifický přepínač `-g parsing`, který usnadňuje hledání výskytu syntaktické chyby pomocí postupných výpisů úspěšně zpracovaných pravidel na standardní chybový výstup.

Aplikace také využívá knihovnu `boost::program_options` ke zpracování argumentů příkazové řádky. Jelikož nejnovější verze knihovny v době psaní aplikace (1.49) obsahovala chybu, jejíž důsledkem nelze v tzv. long options přepínačů používat pomlčku mezi slovy (např. `--architecture-file`), je tato pomlčka poněkud nestandardně nahrazena podtržítkem (`-architecture_file`).

5.2 Princip činnosti interpretu

Interpretací zdrojového kódu jazyka popisující architekturu rozumíme vykonávání funkce `reset` definované v tomto zdrojovém kódu. Jelikož tato funkce definuje chování architektury, její provádění simuluje činnost dané architektury. Během vykonávání interpret zaznamenává události jako je čtení, zápis či volání instrukcí na standardní výstup. Uživatel má při spouštění aplikace možnost pomocí argumentů příkazové řádky volit řadu nastavení, která jsou detailně popsaná v kapitole 5.8.

Jednotlivé fáze činnosti interpretu popisují níže uvedené body.

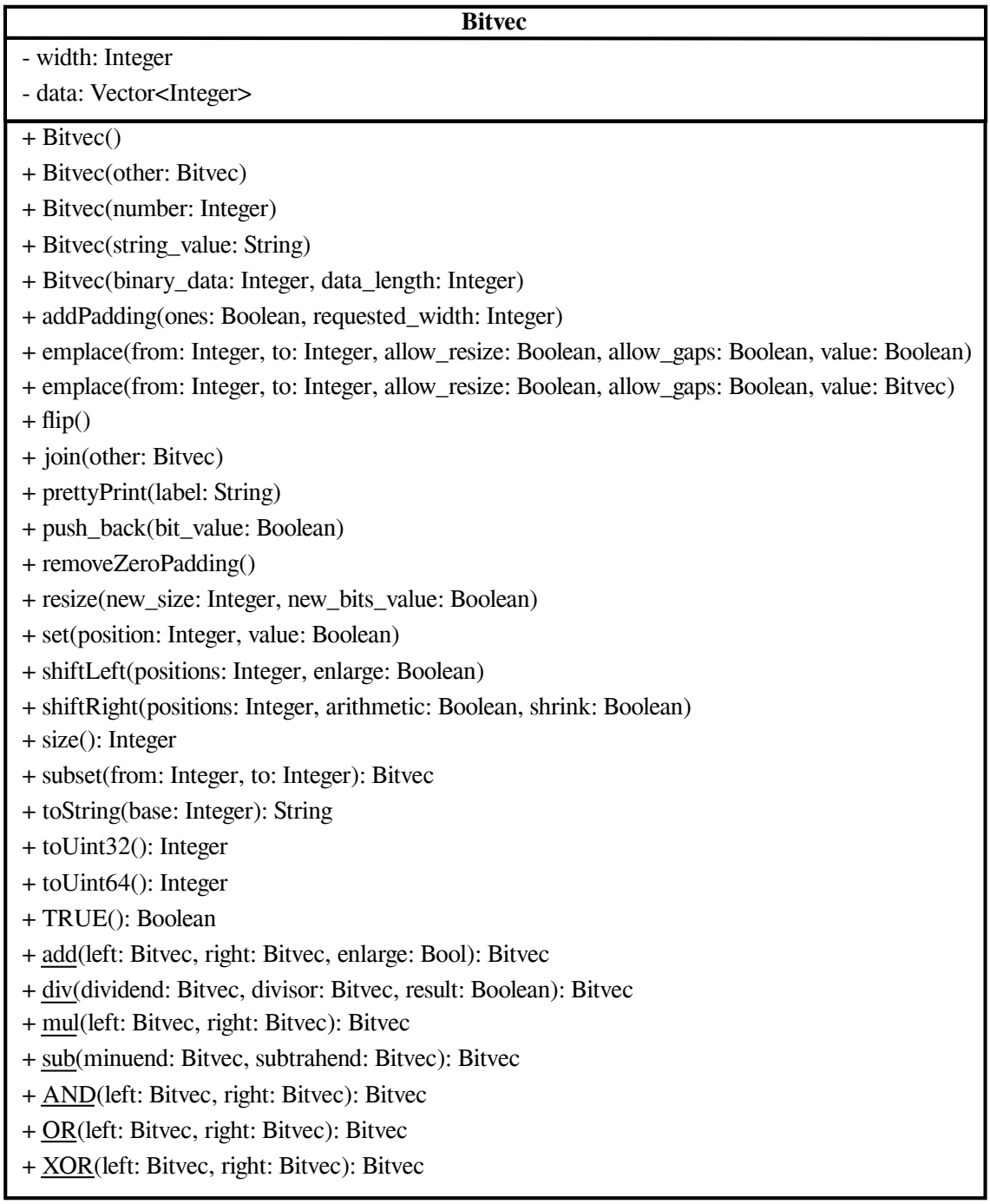
1. Zpracování argumentů příkazové řádky
2. Zpracování zdrojového souboru s popisem architektury, což obsahuje:
   a) Syntaktickou analázu, vytvoření všech entit a funkcí
   b) Vytvoření vyhledávacích datových struktur potřebných k dekódování instrukcí
3. Načtení binárních dat či instrukcí do paměťových entit a to jednou ze dvou možností:
   a) přímé načtení binárního souboru do paměti
   b) načtení kódu zapsaném v jazyku symbolických instrukcí, převedení tohoto kódu do binární podoby a následné načtení binárních dat do paměti
4. Načtení inicializačních a ukončovacích podmínek, jsou-li uživatelem specifikovány
5. Spuštění simulace chování architektury a analýza výstupu

5.3 Datový typ Bitvec

Pro reprezentaci nejen paměťových elementů, ale i výrazů jazyka, bylo potřeba zvolit vhodný datový typ. Jelikož jediným skutečným datovým typem jazyka je binární vektor, požadavky na vhodný datový typ byly následující:

- schopnot rozlišovat hodnoty jednotlivých bitů
- libovolně dlouhá délka
- schopnost provádět základní logické a aritmetické operace
- dynamičnost (délka a hodnota vektoru není známa během překladu)
  - z tohoto důvodu nelze použít `std::bitset` ze standardních knihoven C++

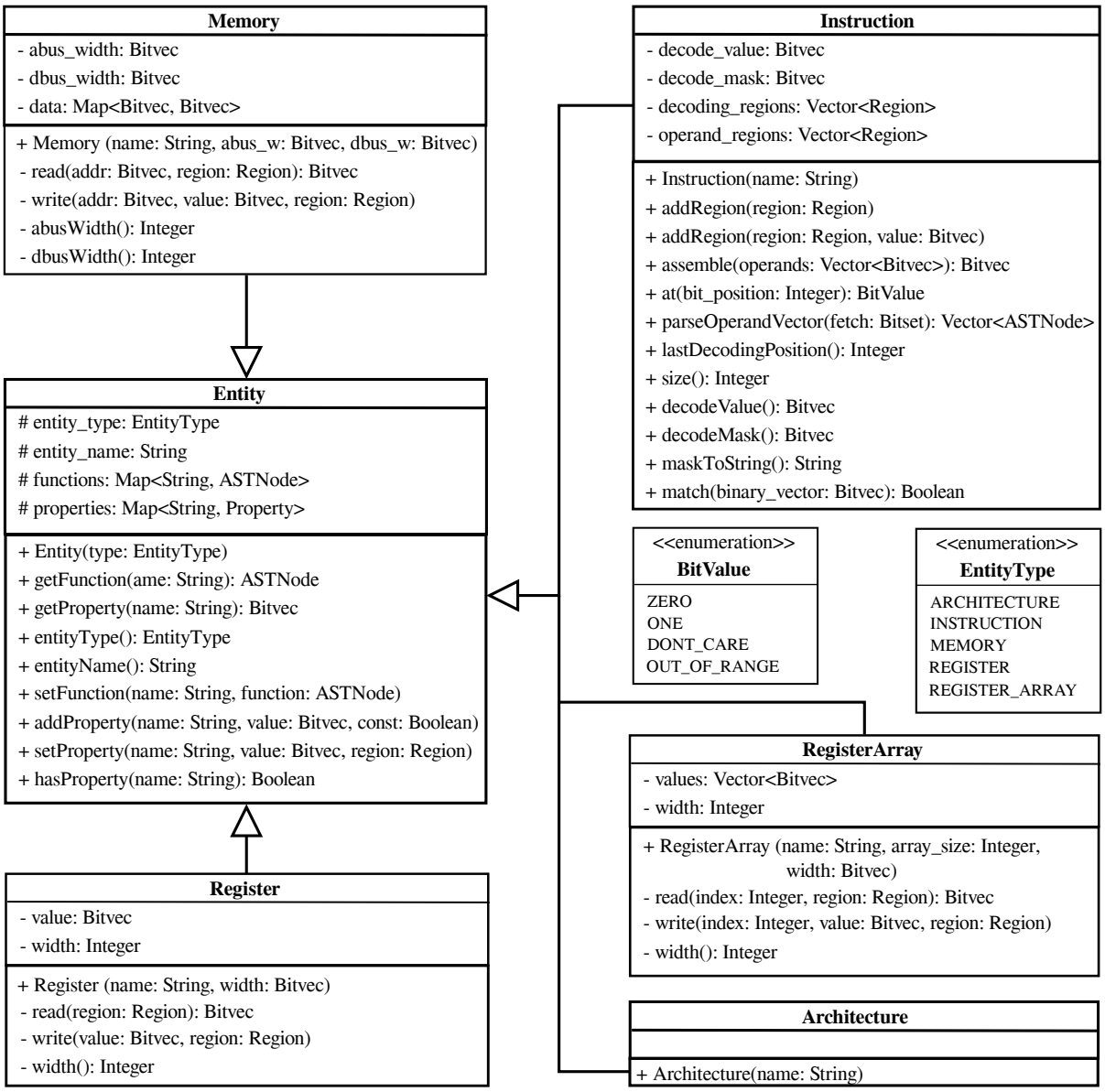
Nejvhodnějším kandidátem se dlouhou dobu zdál být `boost::dynamic_bitset` [15], ale jelikož podporoval jen mizivé množství pro mě potřebné funkcionality, rozhodl jsem se nakonec implementovat vlastní datový typ, který jsem nazval `Bitvec`. Kvůli zachování konzistence je tento datový typ použit i u proměnných a konstant. Na obrázku 5.1 je diagram třídy `Bitvec`.



Obrázek 5.1: Třída `Bitvec`

5.4 Paměťové entity

Na obrázku 5.2 je popsána hierarchie entit. Jedná se o třídy `Architecture`, `Instruction`, `Register`, `RegisterArray` a `Memory`. Všechny zmíněné třídy jsu potomky třídy `Entity`, tudíž je možno jim přiřazovat konstanty, proměnné a funkce. Maximální velikost bitové šířky registru i šířky adresových a datových sběrnic je implementačně omezena na hodnotu 2^{32}. Následkem toho není možno vytvořit registr ani buňku paměti větší než 512MiB. Maximální počet registrů v registrovém poli je opět 2^{32}.



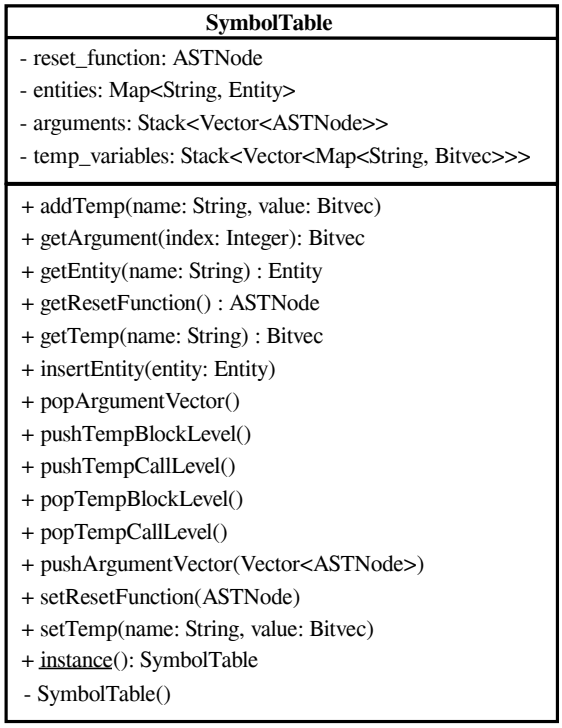
Obrázek 5.2: Hierarchie entit architektury

5.5 Funkce

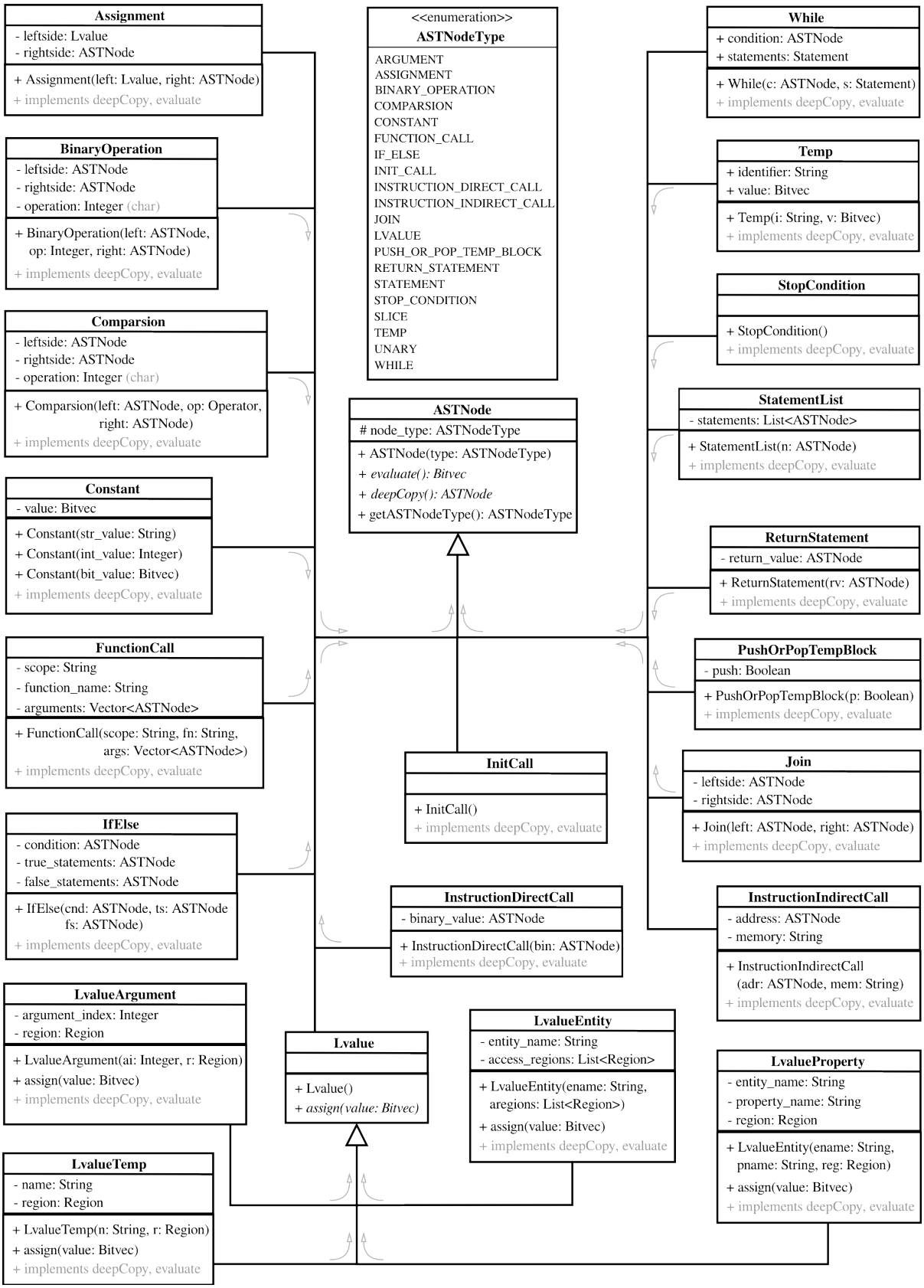
Veškeré chování (tj. funkce a těla instrukcí) je implementováno pomocí abstraktních syntaktických stromů (dále jen *AST*). Každý uzel takového stromu musí být instancí třídy, která implementuje rozhraní abstraktní třídy `ASTNode`. Každý takový objekt implementuje metodu `evaluate()`, která v případě, že to specifický uzel vyžaduje, vrací po vyhodnocení hodotu typu `Bitvec`. Proběhne-li syntaktická analýza bez komplikací, pro každou funkci pak existuje v tabulce symbolů jeden strom tvořený ze specifických uzlů odvozených od `ASTNode`. Vykonání samotné funkce je potom zavolání funkce `evaluate()` na kořenový uzel dané funkce. Přehled všech možných typů uzlů AST ukazuje obrázek 5.3.

5.6 Tabulka symbolů

Tabulka symbolů, implementovaná třídou `SymbolTable` (jde o singleton třídu), která je na obrázku 5.4, obsahuje informace o všech entitách, funkcích, argumentech a lokálních proměnných. Tabulka symbolů je viditelná všem uzlům AST, protože většina z nich vykonává nějakou činnost spjatou s přístupem do ní. Tabulka obsahuje dva separátní zásobníky pro dočasné symboly. Prvním z nich je zásobník argumentů funkcí, který je využíván i pro ukládání operandů instrukcí.



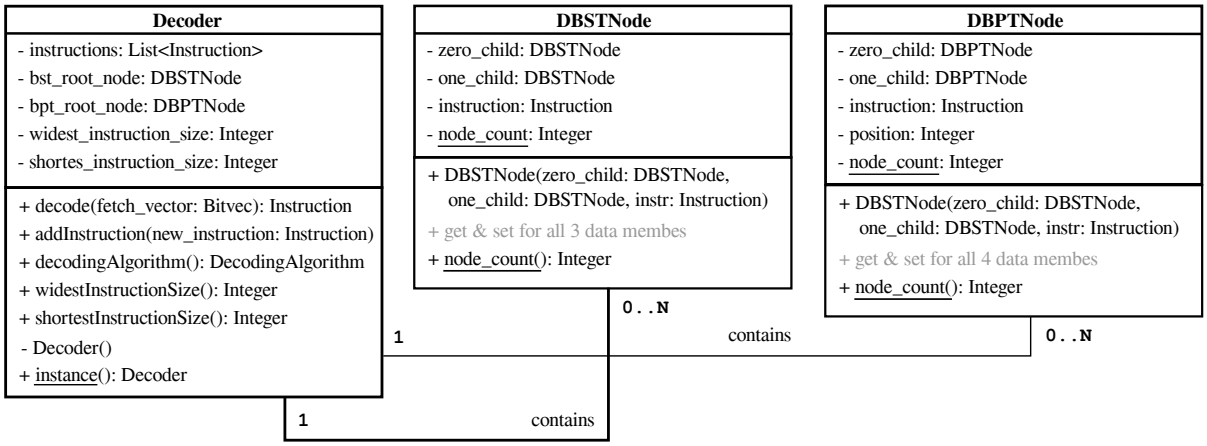
Obrázek 5.4: Tabulka symbolů



Obrázek 5.3: Uzly abstraktního syntaktického stromu

5.7 Dekodér

Dekódování binárních instrukcí má na starosti objekt třídy `Decoder` (opět jde o singleton), který si určitých případech buduje interní stromové struktury, pomocí kterých dekódování provádí. Jelikož `Decoder` zná tři rozdílné dekódovací metody a dvě z nich používají k dekódování stromové struktury, v závislosti na uživatelem zvolené metodě může dekodér vybudovat stromovou strukturu z uzlů jednoho či druhého typu. Dekódovací přístupy jsou detailněji popsány v 5.9. Obrázek 5.4 pouze orientačně popisuje třídu `Decoder` a typy uzlů stromů používaných k dekódování.



Obrázek 5.4: Diagram tříd zobrazující `Decoder` a typy jeho uzlů

5.8 Spouštění aplikace

Jelikož jde o (zatím) konzolovou aplikaci, při spuštění je třeba nastavit několik parametrů potřebných pro samotný běh. Zpracování těchto parametrů má na starosti instance třídy `Settings` (třetí a poslední singleton v celé aplikaci). Třída `Settings` je popsána obrázkem 5.5. V následujících podkapitolách budou detailně popsány všechny možné spouštěcí parametry. V notaci popisující syntax parametrů příkazové řádky

5.8.1 Parametr `-a, --architecture_file`

`-a <soubor>`

Jedná se o jediný povinný parametr, který specifikuje cestu ke zdrojovému souboru, v němž je popsána architektura, kterou chceme simulovat.

5.8.2 Parametr `-b`, `--binary_input`

`-b soubor adresa název_paměti [soubor adresa název_paměti ...]`

Před spuštěním simulace zařídí, aby data z binárního souboru `soubor` byla nahrána do paměti, která se v tabulce symbolů vyskytuje pod jménem `název_paměti` a to počínaje adresou `adresa` v této paměti. Přesahují-li binární data rámec paměti (ať už z důvodu příliš vysoké adresy, nebo nadměrné velikosti souboru), jde o selhání a aplikace je ukončena.

5.8.3 Parametr `-d`, `--decoding_method`

`-d bstree | bptree | brute`

Jako dekódovací metodu zvolí buďto binární vyhledávací strom (`bstree`), binární poziční strom (`bptree`) nebo vyhledávání pomocí seznamu (`brute`). Výchozí hodnota je `brute`. Tyto metody jsou popsány v 5.9.

5.8.4 Parametr `-g`, `--debug_options`

`-g [assembling] [parsing] [astcall] [decode]`

Nastavuje možnosti vypisování ladících zpráv na standardní chybový výstup. Možnosti lze navzájem kombinovat. Volba `assembling` zapříčiní, že při spuštění aplikace s parametrem `-s` budou instrukce i data před zapsáním do paměti vypisovány a lze tak kontrolovat, zda se nahrávají správně. Volba `parsing` je vhodná zejména pro hledání syntaktických chyb, neboť provádí výpisy během syntaktické analýzy. Volba `astcall` provádí výpisy při vyhodnocování jednotlivých uzlů AST a s její pomocí lze zjistit, z jakých uzlů se strom skládá, zda se volají ve správném pořadí a jaké informace jednotlivé uzly nesou. Volba decode vytiskne informace o struktuře dekódovacího stromu, existuje-li.

5.8.5 Parametr `-h`, `--help`

`-h`

Vypíše nápovědu na standardný výstup.

5.8.6 Parametr `-i`, `--initialisation`

`-i soubor`

Nastaví `soubor` pro inicializaci stavu architektury. Tento soubor obsahuje příkazy v totožné syntaxi, jakou má navržený jazyk. Příkazy jsou vykonány v momentě, kdy je během vykonávání chování architektury vyhodnocena funkce `INIT()`.

5.8.7 Parametr `-l`, `--log_options`

`-l [read [READ_FORMAT]] [write [WRITE_FORMAT]] [execute [EXECUTE_FORMAT]]`

Nastavuje možnosti výpisů událostí na standardní výstup. Určuje, které ze základních tří druhů událostí budou vypisovány a v jakém formátu.

Volbou `read` aktivujeme výpis při čtení z paměťových entit. `READ_FORMAT` je "uvozovkami ohraničený" textový řetězec, ve kterém jsou následující klíčová slova převedena na následující informace:

- `%name` jméno entity, ze které je čtena hodnota
- `%type` typ entity, ze které je čtena hodnota
- `%region` region, který je použit pro čtení hodnoty
- `%readval` přečtená hodnota (může být podmnožinou originální hodnoty)
- `%origval` celá originální hodnota

Není-li explicitně specifikován `READ_FORMAT`, použije se výchozí hodnota, která je stejná, jako kdyby byl zadán parametr

`-l read "%name%region => %readval"`.

Volbou `write` aktivujeme výpis při zápisu do paměťových entit. `WRITE_FORMAT` je řetězec, ve kterém jsou následující klíčová slova převedena na následující informace:

- `%name` jméno entity, do které je hodnota zapisována
- `%type` typ entity, do které je hodnota zapisována
- `%origval` hodnota entity před zápisem
- `%newval` hodnota entity po zápisu
- `%region` region použitý pro zápis
- `%assignval` region použitý pro zápis

Výchozí `WRITE_FORMAT` odpovídá spuštění parametru

`-l write "%name%region <= %assignval"`.

Volbou `execute` aktivujeme výpis události vykonávání instrukce (přesněji jde o okamžik mezi rozpoznáním a začátkem vykonáváním instrukce). `EXECUTE_FORMAT` nahrazuje následující řetězce:

- `%iname` jméno instrukce, která bude vykonána
- `%ibin` binární podoba instrukce

Výchozí `EXECUTE_FORMAT` je

`-l execute "----------\n%iname (%ibin)"`.

5.8.8 Parametr `-n`, `--numeric_base`

`-n základ`

Specifikuje, základ číselné soustavy, pomocí které budou prováděny výpisy hodnot při čtení a zápisu. Platným základem jsou čísla od 2 do 16. Výchozí základ je 10.

5.8.9 Parametr `-s`, `--assembly_input`

`-s soubor název_paměti [soubor název_paměti ...]`

Zpracuje `soubor` obsahující kód jazyka symbolických instrukcí a instrukce v něm obsažené převede do binárná podoby, kterou nahrává do paměti `název_paměti`. Jde o alternativu k parametru `-b`. Syntax jazyka symbolických instrukcí je natolik jednoduchá, že na její vysvětlení postačí následující příklad.

```
@7        ; Adresa v paměti, na kterou se začnou ukládat binární data.
add 14    ; Vytvoř binární podobu instrukce 'add' (instrukce jménem add
          ; musí být definována ve zdrojovém souboru architektury) a na
          ; místo jejího jediného operandu vlož hodnotu 14. Takto vzniklou
          ; instrukci vlož do paměti na adresu 7.
0b1001    ; Vlož konstantu 0b1001 do paměti na adresu 8.
xor 2, 3  ; Vytvoř binární podobu 'xor', na místo prvního operandu vlož 2,
          ; na místo druhého operandu vlož 3 a vzniklou instrukci vlož do
          ; paměti na adresu 9.
@48       ; Nastav čítač adresy na hodnotu 48.
0b111101  ; Vlož konstantu 0b111101 na adresu 48.
add 7     ; Vytvoř instrukci add s operandem 7 a vlož ji na adresu 49.
```

Zdrojový soubor asembleru může obsahovat libovolné množství takovýchto souvislých bloků instrukcí. Všechny operandy je nutno zadávat číselně.

5.8.10 Parametr `-t`, `--stop_conditions`

`-t soubor all | any`

Nastaví `soubor` jako soubor pro kontrolu ukončovacích podmínek. Ukončovací podmínky jsou takové podmínky, jejichž naplnění povede k ukončení simulace. Modifikátor `any` znamená, že k ukončení dojde po naplnění alespoň jedné z podmínek, zatímco modifikátor `all` zajistí, že k ukončení simulace dojde teprve tehdy, když jsou splněny všechny podmínky zároveň. Formát souboru specifikující podmínky ozřejmuje následující příklad.

```
(pc <= 50)
(acc != 47)
(ram[4..5] != 0x3f3f)
```

Soubor obsahuje sekvenci porovnávacích výrazů uzavřených do jednoduchých závorek. Tyto podmínky jsou, podle zvolené vyhodnocovací politiky (`all` nebo `any`), vyhodnocovány pokaždé, když po spuštění funkce `reset` dojde k vyhodnocení výrazu `RUN`.

5.8.11 Parametr `-u, --undefined_values`

`-u zero | failure`

Specifikuje, jak zacházet s nedefinovanými hodnotami. Ačkoliv registry mají vždy implicitně nulovou hodnotu, u paměti lze volbou `failure` vynutit, že přístup k nedefinovaným hodnotám (tj. hodnotám, do kterých nebyla explicitně přiřazena hodnota) je nepřijatelný a ukončí simulaci. Výchozí hodnota je `zero`.

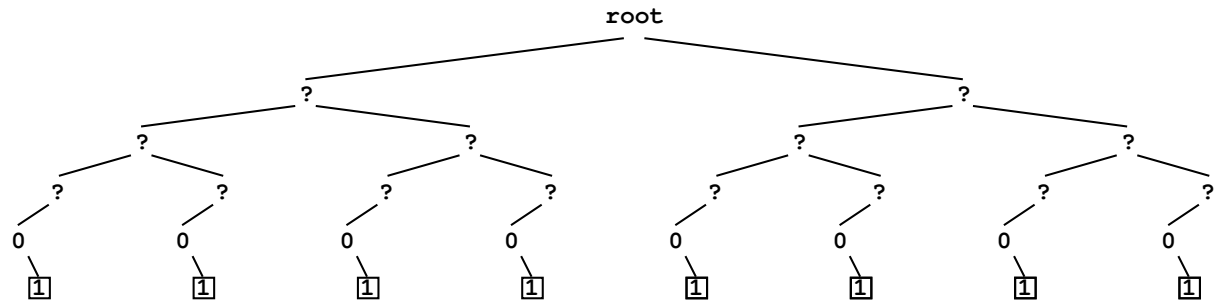
5.9 Dekódovací metody

5.9.1 Binární vyhledávací strom

Dekódování pomocí binárního vyhledávacího stromu (dále jen *BST*) lze vynutit spuštěním aplikace s přepínačem `-d bstree`. Kořen dekódovacího BST je uložen v dekodéru. Následuje detailní popis konkrétní podoby používaného BST.

Každý *uzel BST* (lze jej vidět na obrázku 5.4 pod názvem `DBSTNode`) nese tři informace. Jedná se o *ukazatel na instrukci*, *ukazatel na nulového potomka* a *ukazatel na jedničkového potomka*. Bezprostředně po spuštění aplikace ještě není známa binární podoba jednotlivých instrukcí a proto je v dekodéru vytvořen prázdný kořenový uzel, jehož všechny tři hodnoty jsou inicializovány na prázdnou hodnotu (nullptr). Je zřejmé, že pro samotné dekódování je nutné nejdříve dekódovací BST vytvořit. Tento proces se odehrává již během syntaktické (resp. sémantické) analýzy a to s každou úspěšně zpracovanou instrukcí.

Z pohledu dekódování nás u každé instrukce zajímá *maska instrukce*, která zachycuje její binární podobu. Je-li instrukce definovaná ve zdrojovém kódu úspěšně zpracována (tzn. nevyskytuje se v ní syntaktická chyba ani sémantická chyba), má dekodér k dispozici její masku pokusí se tuto instrukci přidat do BST.



Obrázek 5.5: Struktura BST po vložení instrukce s maskou `???10???`

Přidání jedné instrukce do BST popisuje algoritmus 5.1.

```
Do zásobníku AP vlož dvojici (kořen, 0).
Dokud není AP prázdný
| Do NODE, POSITION vlož hodnoty z vrcholu AP
| Odstraň prvek z vrcholu AP
| Dokud POSITION < INSTRUCTION.size()
| | Je-li NODE.getInstruction() neprázdný
| |   Selhání (nejednoznačné instrukce)
| | Je-li INSTRUCTION[POSITION] bit operačního kódu
| | | Je-li INSTRUCTION[POSITION] bit 0
| | | | Je-li POSITION == INSTRUCTION.lastDecodingPosition()
| | | | | Je-li NODE.getZeroChild() neprázdný
| | | | |   Selhání(instruke způsobuje nejednoznačnost)
| | | | | Jinak
| | | | |   Vytvoř uzel LAST_NODE obsahující instrukci INSTRUCTION
| | | | |   NODE.setZeroChild(LAST_NODE)
| | | | Jinak
| | | |   Neexistuje-li uzel NODE.getZeroChild(), vytvoř jej
| | | |   NODE = NODE.getZeroChild()
| | | Jinak
| | |   Je-li POSITION == INSTRUCTION.lastDecodingPosition()
| | |   | Je-li NODE.getOneChild() neprázdný
| | |   |   Selhání(instruke způsobuje nejednoznačnost)
| | |   | Jinak
| | |   |   Vytvoř uzel LAST_NODE obsahující instrukci INSTRUCTION
| | |   |   NODE.setOneChild(LAST_NODE)
| | |   Jinak
| | |     Neexistuje-li uzel NODE.getOneChild(), vytvoř jej
| | |     NODE = NODE.getOneChild()
| | Jinak
| |   Je-li POSITION == INSTRUCTION.lastDecodingPosition()
| |   | Je-li NODE.getOneChild() nebo NODE.getZeroChild() neprázdný
| |   |   Selhání(nejednoznačnost)
| |   | Jinak
| |   |   Vytvoř uzel LAST_NODE obsahující instrukci INSTRUCTION
| |   |   NODE.setZeroChild(LAST_NODE)
| |   |   NODE.setOneChild(LAST_NODE)
| |   Jinak
| |     Neexistuje-li uzel NODE.getOneChild(), vytvoř ho
| |     Vlož do AP dvojici (NODE.getOneChild(), POSITION + 1)
| |     Neexistuje-li uzel NODE.getZeroChild(), vytvoř ho
| |     NODE = NODE.getZeroChild()
| | POSITION = POSITION + 1
```

Algoritmus 5.1: Vkládání instrukce do BST

Kdyby instrukční maska obsahovala pouze operační kód (tedy `1`, `0` a nikoliv `?` bity), byl by takto vzniklý strom vždy optimální vyhledávací strukturou. Bity operačního kódu ale mohou situaci značně komplikovat. Pro každý bit operandu vzniká pro vkládanou instrukci nová, alternativní cesta, pokračující až do terminálního uzlu obsahující instrukci. Nazveme-li počet bitů operandů od začátku instrukce až po poslední bit operačního kódu N, počet alternativních cest je právě 2^N. Je tedy evidentní, že v případě některých instrukcí resp. instrukčních sad může být tento způsob zcela nepřijatelný kvůli extrémnímu nárůstu prostorové složitosti.

Na obrázku 5.5 vidíme ilustraci tohoto problému. Do prázdného dekodéru je vložena jediná instrukce, jejíž maska je `???10???`. Kvůli pouhým dvěma bitům operačního kódu bylo vytvořeno 8 alternativních cest a celý strom je složen z 31 uzlů. Uzly označené čtverečkem jsou terminální uzly obsahující instrukci (konkrétně odkaz na entitu `Instruction`).

5.9.2 Binární poziční strom

Binární poziční strom (jedná se o mnou vymyšlený termín, dále jen *BPT*) je dalším možným přístupem k dekódování instrukcí. Lze jej vynutit spuštěním aplikace s přepínačem `-d bptree`. Struktura *uzlu BPT* je na obrázku 5.4 pod názvem `DBPTree`. Kořen BPT je opět uložen v dekodéru. Princip dekódování tímto přístupu spočívá v tom, že při traverzování binárního stromu se pro vybírání cesty nerozhodujeme podle aktuálně zpracovávaného bitu binárního vektoru, ale podle pozice, která je pevnou součástí právě zpracovávaného uzlu.

Konstrukce BPT je odlišná od konstrukce BST. Vyžaduje totiž znalost všech instrukcí, resp. všech instrukčních masek, a proto k ní dochází až po ukončení syntaktické (resp. sémantické) analýzy. Před samotným budováním stromové struktury je potřeba explicitně zkontrolovat jednoznačnost (tj. rozlišitelnost na základě operačního kódu) všech instrukcí definovaných v architektuře. Tento přístup má jednu zcela zásadní nevýhodu. Je prakticky nepoužitelný u instrukčních sad s různou délkou instrukcí. Může se totiž stát, že je-li dekódovaná instrukce příliš krátká, rozhodovací pozice uzlu je mimo tuto instrukci a není možno se rozhodnout, kterým směrem ve vyhledávání pokračovat.

Při budování BPT je každém kroku algoritmu 5.2 pomocí funkce 5.3 hledána taková pozice, která seznam instrukcí příslušející k danému kroku rozdělí do dvou co nejvíce disjunktních tříd. Tato pozice se stane rozlišovací pozicí vytvořeného uzlu. V každém kroku se počet instrukcí pro následující krok sníží minimálně o 1 a maximálně o polovinu počtu instrukcí v seznamu. Tento přístup vylučuje exponenciální nárůst prostorové složitosti, který je nedostatkem BST přístupu, a dokáže se přizpůsobit i instrukcím s maskami obsahujícími velký počet bitů operandů (ba naopak, velký počet operandů může tvorbu BPT usnadnit).

Samotné dekódování je jednoduché. Podle hodnoty binárního vektoru na pozici určené uzlem traverzujeme až do terminálního uzlu. Narozdíl od BST však nalezení terminálního uzlu neznamená, že byla rozpoznána instrukce. Znamená to, že hodnota dekódovaného binárního vektoru buďto odpovídá instrukci v tomto uzlu, nebo neodpovídá žádné instrukci. Zbývá tedy porovnat masku instrukce v uzlu s binárním vektorem a teprve je-li výsledkem tohoto porovnání shoda, byla úspěšně dekódována instrukce.

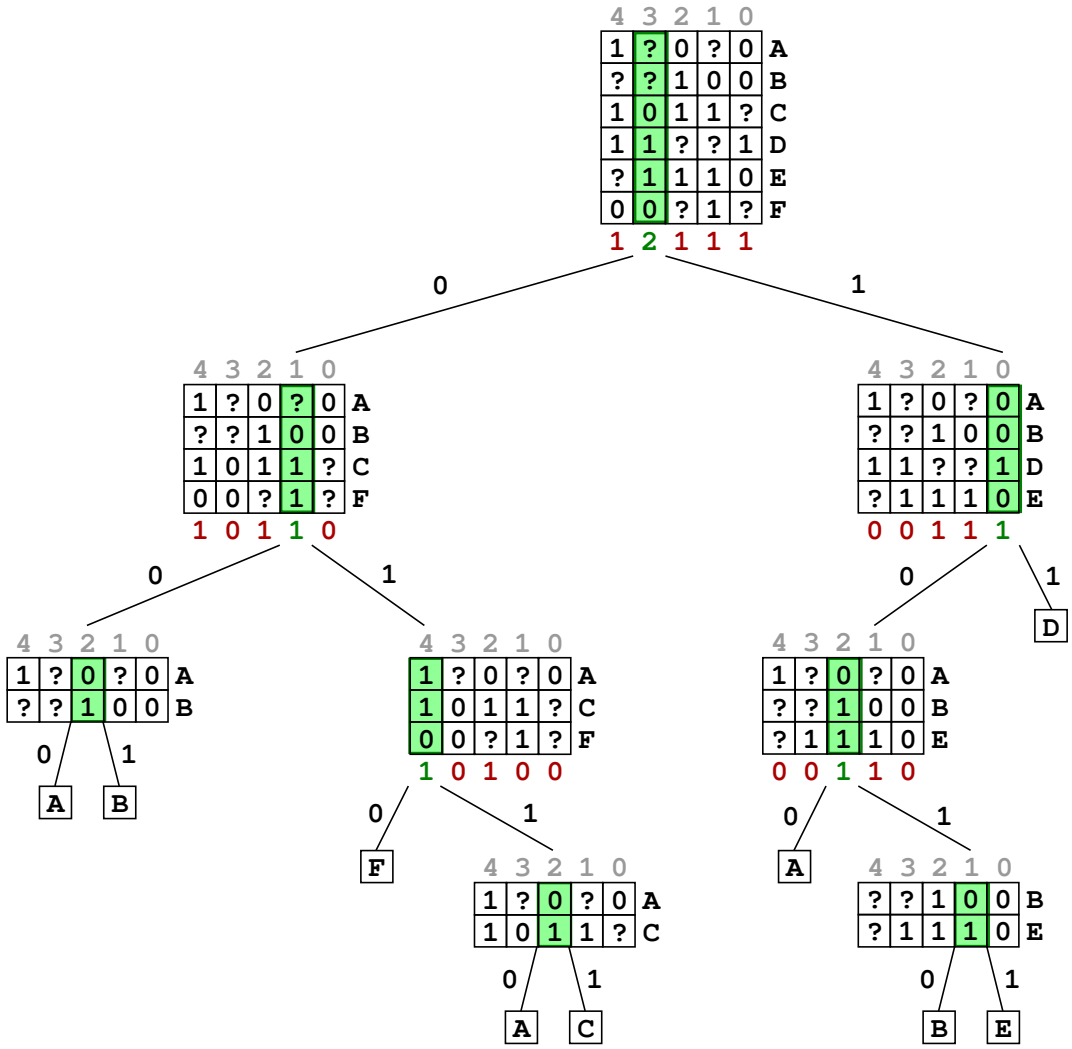
```
Do zásobníku STAGES vlož dvojici (kořen, seznam_všech_instrukcí)
Dokud není STAGES prázdný
| Do NODE, INSRUCTION_LIST vlož hodnoty z vrcholu STAGES
| Odstraň prvek z vrcholu STAGES
| Do DIST_POSITION vlož výsledek funkce distPosition(INSTRUCTION_LIST)
| NODE.position = DIST_POSITION
| Vytvoř prázné seznamy instrukcí ZERO_LIST a ONE_LIST
| Do seznamu ZERO_LIST vlož ty instrukce z INSTRUCTION_LIST, které mají
|   na pozici DIST_POSITION hodnotu 0 nebo ?
| Má-li ZERO_LIST velikost 1
| | Vytvoříme uzel TERM_NODE, který obsahuje instrukci ze ZERO_LIST
| | NODE.setZeroChild(TERM_NODE)
| Jinak
|   Vytvoříme prázný uzel NONTERM_NODE
|   NODE.setZeroChild(NONTERM_NODE)
|   Do STAGES vložíme dvojici (NONTERM_NODE, ZERO_LIST)
| Do seznamu ONE_LIST vlož ty instrukce z INSTRUCTION_LIST, které mají
|   na pozici DIST_POSITION hodnotu 1 nebo ?
| Má-li ONE_LIST velikost 1
| | Vytvoříme uzel TERM_NODE, který obsahuje instrukci z ONE_LIST
| | NODE.setOneChild(TERM_NODE)
| Jinak
|   Vytvoříme prázný uzel NONTERM_NODE
|   NODE.setOneChild(NONTERM_NODE)
|   Do STAGES vložíme dvojici (NONTERM_NODE, ONE_LIST)
```

Algoritmus 5.2: Budování BPT

```
FUNKCE distPosition(INSTRUCTION_LIST):
Vytvoř proměnné BEST_POS = 0, BEST_RATING = 0, BEST_DC = MAX
Pro každé N od 0 do poslední pozice nejdelší instrukce v INSTRUCTION_LIST
| Do proměnné ZERO_CNT vlož počet výskytů hodnoty 0 na pozici N ve všech
|   instrukcích v INSTRUCTION_LIST
| Do proměnné ONE_CNT vlož počet výskytů hodnoty 1 na pozici N ve všech
|   instrukcích v INSTRUCTION_LIST
| Do proměnné DC_CNT vlož počet výskytů hodnoty ? a OUT_OF_RANGE (délka
|   instrukce je <= N) na pozici N ve všech instrukcích v INSTRUCTION_LIST
| Do proměnné RATING ulož menší z hodnot ZERO_CNT, ONE_CNT
| Je-li RATING > BEST_RATING
|   BEST_POS = N
|   BEST_DC = DC_CNT
| Je-li RATING == BEST_RATING
|   Je-li DC_CNT < BEST_DC
|     BEST_POS = N
|     BEST_DC = DC_CNT
Jako výsledek funkce vrať BEST_POS
```

Algoritmus 5.3: Funkce `distPosition`

Velmi názorně objasňuje sestrojování a zároveň i výslednou strukturu BPT obrázek 5.6. Jde o výsledek sestrojování BPT ze šesti instrukcí `A`, `B`, `C`, `D`, `E` a `F`. Tabulky na místech uzlů stromu představují seznam instrukcí právě zpracovávaného uzlu. V tabulce je zeleně je vyznačena pozice, která byla algoritmem 5.3 vyhodnocena jako nejvhodnější pro rozdělení na dvě třídy instrukcí. Zeleným písmem je vyznačen výsledek této hodnotící funkce.



Obrázek 5.6: Sestrojování BPT

5.9.3 Vyhledávání seznamem

Jde o výchozí metodu (přepínač `-d brute`). Tato metoda je ze všech tří nejprimitivnější, nejméně efektivní (lineární časová složitost), ale zato nemůže selhat.

Její princip je velmi jednoduchý. Lineární seznam instrukcí, u kterého bylo ověřeno, že neobsahuje nejednoznačné instrukce, prochází od začátku do konce a hledá shodu s instrukční maskou jednotlivých instrukcí. Projde-li celým seznamem beze shody, dekódování selhalo.

5.9.4 Srovnání dekódovacích metod

Každá ze tří možných dekódovacích metod může ve specifických případech být výhodnější, než ostatní.

Metoda BST má oproti BPT dvě výhody. První výhodou je menší velikost uzlu. V případě architektury s instrukční sadou, ve které všechny instrukce mají stejně dlouhý souvislý operační kód, který začíná na pozici 0, bude struktura stromu u obou metod totožná. BST ale v paměti zabere nepatrně menší prostor, protože jeho uzly nenesou pozici. Druhou výhodou je, že při vyhledávání pomocí BST je počet traverzačních kroků (tedy časová složitost) omezen šířkou nejdelší instrukce. To neplatí v případě BPT, kdy nelze vyloučit, že jedna pozice bude rozhodovací pozicí vícekrát. Hodláme-li simulovat architekturu se souvislým operačním kódem začínajícím na pozici 0, je vhodější zvolit algoritmus `bstree`.

Metoda BPT má nespornou výhodu v případě instrukčních sad s nesouvislým operačním kódem konstantní délky, protože pozice operandů při tvorbě stromu velmi často vynechává. Jakkoliv je výhodou potenciálně menší velikost vyhledávací stromové struktury, daní za tuto výhodu je zvýšená časová složitost, neboť počet kroků potřebných k dosažení terminálního uzlu může být větší, než je počet bitů instrukce.

Vyhledávání seznamem je časově nejsložitější, ale zvládá i instrukce s různými délkami a navíc má nulovou prostorovou složitost, protože seznam, který prohledává, je pevnou součástí dekodéru.

5.9.5 Proměnlivá délka instrukcí

V kapitole 4.1 jsem zmínil, že v jazyku existují dvě odlišné konstrukce pro dekódování a následné vykonání instrukcí. Nyní popíšu jejich rozdíl z praktického hlediska tak, jak dekódování řeší interpret.

U varianty `execute(výraz)` probíhá dekódování intuitivně. Dekodéru je poslán binární vektor odpovídající hodnotě výrazu, který se dekodér pokusí dekódovat pomocí výše popsaných algoritmů. Ať úspěšně či neúspěšně, tímto dekódování končí.

Příkaz `execute(název_paměti, adresa)` předpokládá, že instrukce mohou mít proměnlivou délku. Nejprve se pokusí dekódovat hodnotu `název_paměti[adresa]`. Tato hodnota má bitovou šířku právě jedné buňky paměti. Není-li v této hodnotě rozpoznána instrukce, připojí se k této hodnotě hodnota `název_paměti[adresa + 1]` a opakuje se pokus o dekódování. K původní hodnotě jsou tak postupně z paměti připojovány hodnoty na následujících adresách. Jelikož dekodér zná délku nejdelší instrukce, dokáže vypočítat, kolik buněk paměti má cenu takto připojovat. Teprve je-li tato mez překročena, dekódování selhalo.

6 Ukázka činnosti interpretu

V této kapitole budu ryze prakticky popisovat spouštění interpretu s různým nastavením parametrů. Jelikož by bylo nepraktické psát do této práce mnohdy rozsáhlé výstupy aplikace, doporučuji čtenáři popsané postupy vlastnoručně vyzkoušet a shlédnout tak výstupy na vlastní oči.

6.1 Hypotetický počítač č.1

V demonstračním příkladu bude využit jednoduchý *Hypotetický počítač č. 1*, který jsem převzal ze studijní opory předmětu IAS [16]. Jde o architekturu se třemi osmibitovými registry, kterými jsou ukazatel instrukcí (`ireg`), instrukční registr (`ir`) a střadač (`acc`). Paměť je adresována 5 bity a velikost datové sběrnice je 1B. Instrukční sada se skládá z osmi instrukcí. Všechny instrukce mají operační kód o délce 3 na pozicích `[5..7]`. Zdrojový kód této architektury je uveden v příloze B. Testovací soubory, které budu v této kapitole zmiňovat, se nacházejí na přílohovém CD v kořenovém adresáři ve složce `app`.

6.1.1 Nahrávání dat do paměti

Před samotným spuštěním je nutné připravit si binární data, která budou uložena do paměti. Vhodnou demonstrací použití, a zároveň testem, bude pokusit se totožná binární data nahrát do paměti nejdříve pomocí skutečného binárního souboru a poté pomocí kódu jazyka symbolických instrukcí. K tomu použijeme soubory testovací soubory `mem/ias1.bin` a `mem/ias1.asm`.

Nejprve spustíme aplikaci s parametry `-a arc/ias1.arc -b mem/ias1.bin ram 0 -g assembling`. Na standardní chybový výstup budou vypsány hodnoty ukládané do paměti. Poté spustíme aplikaci znovu s parametry `-a arc/ias1.arc -s mem/ias1.asm ram -g assembling` a pomocí ladících výpisů se přesvědčíme, že jsou do paměti nahrávána totožná data. Zároveň na standardním výstupu sledujeme, že se i při běhu skutečně vykonávají tytéž instrukce.

6.1.2 Velikost dekódovacích stromů

I na takto směšně jednoduché architektuře lze demonstrovat, jak neefektivní může být dekódovací algoritmus BST. Spustíme-li aplikaci s parametry

```
-a arc/ias1.arc -b mem/ias1.bin ram 0 -g decode -d bstree,
```

na standardním chybovém výstupu vidíme, že strom obsahuje 511 uzlů a zaujímá prostor v řádech kilobajtů (při spuštění aplikace na architektuře s 64 bitovými ukazateli je to 12264B). Jestliže spustíme ten stejný příkaz, ale namísto `bstree` zvolíme `bptree`, najednou máme 15 uzlů zabírajících 420B paměti.

6.1.3 Ukončovací podmínky a inicializační příkazy

Jelikož jde o simulátor, často se při jeho používání můžeme snažit zjistit, dojde-li během běhu architektury k naplnění nějaké podmínky. Toho sice jde docílit explicitně přímo výrazy jazyka, tak, jak je tov případě `arc/ias1.arc`, ale to je poněkud nepraktické. Mnohem výhodnější je použít výrazu `RUN`, který se váže k podmínkám specifikovaným přepínačem `-t`. To mimo jiné umožní simulovat např. pomocí dávkových zpracování a hlavně bez zásahu do kódu architektury. Totožné důvody hovoří pro použití inicializačního příkazu `INIT()`. Příklad popisu výše uvedené architektury pozměněného tak, aby kontroloval ukončovací podmínky, je v souboru `arc/ias2.arc`. Ten lze spustit s následujícími parametry:

```
-a arc/ias2.arc -s mem/ias2.asm ram -t cnd/ias2.cnd any
```

Po spuštění vidímě půběh chování instrukcí ukončený hláškou, která tvrdí, že simulace byla ukončena z důvodu naplnění ukončovacích podmínek.

7 Závěr

Cílem této bakalářské práce bylo vytvořit návrh jednoduchého jazyka pro popis instrukčních sad, resp. architektury mikroprocesoru. Dosud existující jazyky pro popis architektur se vyznačují především velkou složitostí a nízkou měrou abstrakce. Mnou vytvořený jazyk je v tomto směru přívětivý. Navíc je velmi intuitivní a většinu konstrukcí lze pochopit na první pohled, neboť jsou často obdobami konstrukcí jiných, časem prověřených jazyků.

Vytvořený interpret jazyka je schopen simulovat činnost počítačové architektury na úrovni popsané navrženým jazykem.

Největší překážkou při tvorbě práce pro mě byla tvorba samotné aplikace, neboť program postupem času začal neúměrně nabírat na objemu kódu a tím značně utrpěla jeho celková kvalita. Krom velkého množství automaticky vygenerovaného kódu se jedná se o necelých 9000 řádků ručně psaného kódu, které bude patrně vhodné pročistit a refaktorovat, než bude možné ve vývoji aplikace pokračovat.

Další vývoj projektu závisí na tom, jak bude tato práce ohodnocena. Bude-li přijata kladně, zůstanu u současného plánu ji časem rozšiřovat. Jelikož dosavadní aplikace je konzolová a při spouštění je třeba zadávat poměrně velké množství parametrů, logicky se nabízí rozšíření o grafické uživatelské rozhraní. Například možnost krokování simulace by jistě zlepšila uživatelskou přívětivost.

Literatura

[1] Kolektiv autorů. *VHDL Analysis and Standardization Group (VASG)* [online]. 2014-3-14 [cit. 2014-3-14]. Dostupný z WWW: <http://www.eda.org/twiki/bin/view.cgi/P1076/WebHome>

[2] Kolektiv autorů. *Verilog Resources* [online]. 2012-6-10 [cit. 2014-3-14]. Dostupný z WWW: <http://www.verilog.com/>

[3] PERRIER, V.: *A look inside electronic systém level (ESL) design* [online]. 2004-3-27 [cit. 2014-3-19]. Dostupný z WWW: <http://www.eetimes.com/document.asp?doc_id=1276969>

[4] ABRAR, S., S.: *Advances in SoC and Processor Modelling Methodologies* [online]. 2009-4-27 [cit. 2014-2-12]. Dostupný z WWW: <www.design-reuse.com/articles/20577/soc-processor-modeling-methodologies.html>

[5] Kolektiv autorů z konzorcia The Open Group. *ArchiMate 2.1 Specification* [online]. 2013-7-4 [cit. 2014-3-15]. Dostupný z WWW: <http://pubs.opengroup.org/architecture/archimate2-doc/toc.html>

[6] IEEE Standards Association. I*EEE Standard for Standard SystemC® Language Reference Manual* [online]. 2012-1-9 [cit. 2014-4-5]. Dostupný z WWW: <http://standards.ieee.org/getieee/1666/download/1666-2011.pdf>

[7] Synopsis Inc. *Synopsis – Accelerating Innovation* [online]. 2014-4-12 [cit. 2014-4-12]. Dostupný z WWW: <www.synopsys.com/Systems/ArchitectureDesign/Pages/ArchitectureModels.aspx>

[8] The ArchC Team. *The ArchC Architecture Description Language v2.0 Reference Manual* [online]. 2007-8-17 [cit. 2014-2-12]. Dostupný z WWW: <*http://archc.lsc.ic.unicamp.br/downloads/Docs/ac_lrm-v2.0.pdf*>

[9] RWTH Aachen University. *The Language for Instruction Set Architectures* [online]. 2014-2-5 [cit. 2014-2-5]. Dostupný z WWW: <http://www.ice.rwth-aachen.de/research/tools-projects/entry/detail/lisa/>

[10] PEES , S., HOFFMAN, A., ZIVOJNOVIC, V., MEYR, H. *LISA – Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures.* Design Automation Conference, str. 933-398, 1999.

[11] Kolektiv autorů. *ARM® and Thumb®-2 Instruction Set Quick Reference Card.* [online] 2006-3 [cit. 2014-3-19]. Dostupné z WWW: <http://infocenter.arm.com/help/topic/com.arm.doc.qrc0001l/QRC0001_UAL.pdf>

[12] Kolektiv autorů. *Intel 64 and IA-32 Architectures Software Developer's Manuals, Vol. 1 Basic Architecture* [online]. 2009-3 [cit. 2014-4-19]. Dostupné z WWW: <http://download.intel.com/design/processor/manuals/253665.pdf >

[13] BROKKEN, B., F.: *Flexc++ V 2.01.00* [online]. 2014-3-16 [cit. 2014-4-18]. Dostupné z WWW: <http://flexcpp.sourceforge.net/>

[14] BROKKEN, B., F.: *Bisonc++ V 4.09.01* [online]. 2014-5-11 [cit. 2014-5-13]. Dostupné z WWW: <http://bisoncpp.sourceforge.net/>

[15] Kolektiv autorů z Boost community. *dynamic_bitset<Block, Allocator>* [online]. 2013-12-13 [cit. 2014-4-17]. Dostupné z WWW: <www.boost.org/doc/libs/1_54_0/libs/dynamic_bitset/dynamic_bitset.html>

[16] HANÁČEK, P., ZBOŘIL, F.: *ASEMBLERY – Studijní opora.* FIT VUT v Brně, 2006-2, skripta do předmětu IAS.

Příloha A

Gramatika navrženého jazyka

Tato příloha obsahuje kompletní formální popis gramatiky navrženého jazyka v BNF notaci. Gramatika je taktéž součástí přiloženého CD a to v souboru `txt/grammar.bnf`.

```
<source_code> ::=
    <resource> <source_code>
  | <function_definition> <source_code>
  | <resource>
  | <function_definition> ;

<resource> ::= <architecture> | <memory> | <register> | <instruction> ;

<function_definition> ::=
    'reset' <ID> '.' <ID> '(' ')' <stat_block>
  | <ID> '.' <ID> '(' <args> ')' <stat_block> ;

<architecture> ::= 'architecture' <ID> <properties> ';' ;

<memory> ::= 'memory' <ID> '(' <exp> ',' <exp> ')' <properties> ';' ;

<register> ::=
    'register' <ID> '(' <exp> ')' ';'
  | 'register' <ID> '[' <exp> ']' '(' <exp> ')' ';'
  ;

<instruction> ::= 'instruction' <ID> '(' <in_regions> ')' <stat_block> ;

<in_regions> ::= <in_region> | <in_regions> <in_region> ;

<in_region> ::= <region> '==' <exp> | <ID> '=' <region> ;

<region> ::=
    '[' <exp> '..' <exp> ']' '@' <region>
  | '[' <exp> '..' <exp> ']'
  | '[' <exp> ']' '@' <region>
  | '[' <exp> ']'
  ;

<regions> ::= <regions> <region> | <region> ;
```

```
<properties> ::= /* nothing */ | '{' <some_properties> '}' ;

<some_properties> ::= <some_properties> <property> | <property> ;

<property> ::= 'const' <ID> '=' <exp> ';' | <ID> '=' <exp> ';' ;

<stat_block> ::= '{' '}' | '{' <statements> '}' ;

<statements> ::= <statements> <statement> | <statement> ;

<statement> ::=
    'temp' <ID> '=' <exp> ';'
  | <lval> '=' <exp> ';'
  | 'if' '(' <comparsion> ')' <stat_block> 'else' <stat_block>
  | 'if' '(' <comparsion> ')' <stat_block>
  | 'while' '(' <comparsion> ')' <stat_block>
  | <call>
  | 'return' <exp> ';'
  | 'INIT' '(' ')'
  ;

<exp> ::= <lval> | <rval> | <NUM> | <bin_exp> | '-' <exp> | 'RUN' ;

<lval> ::= <ID> '.' <ID> <regions>  |  <ID> '.' <ID>  |
           <ID> <regions>           |  <ID>           ;

<rval> ::= <fun_call> ;

<fun_call> ::= <ID> '(' <args> ')' | <ID> '.' <ID> '(' <args> ')' ;

<bin_exp> ::= <exp> '+'  <exp> | <exp> '-'  <exp> | <exp> '*' <exp>  |
              <exp> '/'  <exp> | <exp> '%'  <exp> | <exp> '&' <exp>  |
              <exp> '|'  <exp> | <exp> '^'  <exp> | <exp> '<<' <exp> |
              <exp> '>>' <exp> | <exp> '~>' <exp> ;

<call> ::= <fun_call> | <execute> ;

<execute> ::= 'execute' '(' <ID> ',' <exp> ')' | 'execute' '(' <exp> ')' ;

<comparsion> ::= <exp> '==' <exp> | <exp> '!=' <exp> | <exp> '<=' <exp> |
                 <exp> '<'  <exp> | <exp> '>'  <exp> | <exp> '>=' <exp> ;

<args> ::= /* empty */ | <exp> | <args> ',' <exp> ;
```

Příloha B

Popis architektury č.1

Příklad zápisu architektury Hypotetický počítač č.1. Tento zdrojový kód je k dispozici na přiloženém CD v adresáři `app/arc/ias1.txt`.

```
architecture IAS {
    stop = 0;
};

register ireg(8);
register ir(8);
register acc(8);

memory ram(5, 8);

reset IAS.behavior() {
    while (ias.stop != 1) {
        ir = ram[ireg];
         execute(ir);
    }
}

// [000?????]
instruction NOP ([5..7 == 0]) {}

// [001?????]
instruction LOAD ([5..7 == 1], addr = [0..4]) {
    acc = ram[addr];
    ireg = ireg + 1;
}

// [010?????]
instruction ADD ([5..7] == 2, addr = [0..4]) {
    acc = acc + ram[addr];
    ireg = ireg + 1;
}
```

```
// [011?????]
instruction SAVE([5..7] == 3, addr = [0..4]) {
    ram[addr] = acc;
    ireg = ireg + 1;
}

// [100?????]
instruction NEG([5..7] == 4) {
    acc = -acc;
}

// [101?????]
instruction JMP([5..7] == 5, addr = [0..4]) {
    ireg = addr;
}

// [110?????]
instruction JN([5..7] == 6, addr = [0..4]) {
    if (acc[7] == 1) {
        ireg = addr;
    } else {
        ireg = ireg + 1;
    }
}

// [111?????]
instruction HALT([5..7] == 7) {
    ias.stop = 1;
}
```

Příloha C

Překlad a instalace

Zdrojový kód aplikace se na přiloženém CD vyskytuje v adresáři `src`. K úspěšnému překladu je třeba přilinkovat knihovnou `boost::program_options` (testováno na verzi 1.49) a překladač musí znát cestu ke kořenovému adresáři knihovny `boost`. Pro jistotu byla na CD do složky libs vložen archiv, pomocí něhož lze tato knihovna nainstalovat. Překladač C++ musí podporovat standard C++11.

Aplikace byla primárně vyvíjena a testována pro operační systému Linux (testováno na 64 bitovém Debian Linuxu). Jelikož kromě knihovny boost, která sama sebe deklaruje jako multiplatformní, nebyla použita žádná jiná externí knihovna, měla by aplikace být přeložitelná bez váraznějších změn na většině platformách.

K přeložení aplikace stačí v příkazové řádce zadat v kořenovém adresáři příkaz `make`. Ten vytvoří binární spustitelný soubor ve složce `bin`. Tento soubor se poté spouští s parametry uvedenými v kapitolách této práce a na standardní výstup vypisuje potřebné informace.

Příloha D

Obsah CD

```
/arc/ ... příklady zdrojových souborů
/bin/ ... výsledný spustitelný soubor
/cnd/ ... příklady definice podmínek
/doc/ ... adresář pro generování dokumentace pomocí nástroje Doxyen
/ini/ ... příklady souborů s inicializačními podmínkami
/libs/ ... celá knihovna boost (třeba nainstalovat)
/mem/ ... *.bin a *.asm soubory s obsahy pamětí
/src/ ... zdrojové kódy interpretu
/Makefile ... makefile pro překlad
/doxycfg ... konfigurace pro Doxygen
/xforej02.odt ... zdrojový text bskalářské práce
/xforej02.pdf ... zdrojový text BP převedený do PDF
```